Iterations for Propensity Score Matching in MonetDB

Böhlen, Michael Hanspeter ; Dolmatova, Oksana ; Krauthammer, Michael ; Mariyagnanaseelan, Alphonse ; Stahl, Jonathan ; Surbeck, Timo

Abstract: The amount of data that is stored in databases and must be analyzed is growing fast. Many analytical tasks are based on iterative methods that approximate optimal solutions. Propensity score matching is a technique that is used to reduce bias during cohort building. The main step is the propensity score computation, which is usually implemented via iterative methods such as gradient descent. Our goal is to support efficient and scalable propensity score computation over relations in a column-oriented database. To achieve this goal, we introduce shape-preserving iterations that update values in existing tuples until a fix point is reached. Shape-preserving iterations enable gradient descent over relations and, thus, propensity score matching. We also show how to create appropriate input relations for shape-preserving iterations with randomly initialized relations. The empirical evaluation compares in-database iterations with the native implementation in MonetDB where iterations are flattened.

DOI: https://doi.org/10.1007/978-3-030-54832-2_15

Iterations and Propensity Score Matching in MonetDB

Michael Böhlen[1,2], Oksana Dolmatova[1,2], Michael Krauthammer[1,3], Alphonse Mariyagnanaseelan[1,4], Jonathan Stahl[1,4], and Timo Surbeck[1,4]

[1] University of Zurich, Switzerland
[2] {dolmatova,boehlen}@ifi.uzh.ch
[3] michael.krauthammer@yale.edu
[4] {alphonse.mariyagnanaseelan,jonathan.stahl,timo.surbeck}@uzh.ch

Abstract. The amount of data that is stored in databases and must be analyzed is growing fast. Many analytical tasks are based on *iterative methods* that approximate optimal solutions by refining an initially guessed solution through fixed-point iterations that update the initial values. This paper describes an extension of SQL with *shape-preserving iterations* that update values in existing tuples until a fixed point is reached. We use logistic regression and propensity score matching as a use case to illustrate the implementation of shape-preserving iterations in MonetDB. We show how to create randomly initialized starting points for iterations and how to use the gradient descent technique to compute the logistic regression. The empirical evaluation compares in-database iterations with the native implementation where iterations are flattened.

Keywords: Gradient Descent · Propensity Score Matching · Column Stores · Iterative Methods · Data Science · MonetDB

1 Introduction

In the era of big data analytics many researchers have to deal with constantly growing data sets that must be analyzed with state-of-the-art data science methods that are based on iterative methods. *Propensity score matching* is a statistical technique that estimates the effect of intervention, e.g., a medical treatment. It reduces the bias that exists if we simply compare outcomes among patients that received the treatment versus those that did not. The propensity score can be used to build patient cohorts that have a similar propensity score. It is a numerical approximation that is based on fixed-point iterations and uses gradient descent to iteratively refine the initial values. Since neither gradient descent nor shape-preserving iterations are available in SQL, applications must export, transform, and import data into statistical analysis environments [3,1] to compute propensity scores.

The goal of this paper is to enlarge the range of analytical methods that database systems support by extending SQL with iterative methods that can be applied to data stored in relations. Towards this goal, we extend SQL with *shape-preserving iterations*, which permits in-database propensity score matching. Shape-preserving iterations support methods that repeatedly refine a set of values until a fixed point is reached. Shape-preserving iterations typically start out with a *randomly initialized relation* and we show

how to initialize relations that have the required schema and tuples. One important property of shape-preserving iterations is that they iterate over and return relations with *contextual information* [7,6]. Contextual information guarantees that each relation has a proper schema and includes an attribute with values that allow to identify each cell in a relation. We discuss the adjustments that are required to fully integrate shape-preserving iterations into a column-oriented database and we show the feasibility of our solution by conducting an experimental evaluation of our approach.

Out technical contributions are the following:

- We define *shape-preserving iterations* and integrate these into SQL with only minimal syntactic extensions of SQL.
- We introduce *randomly initialized relations* and show how to define appropriate starting points for shape-preserving iterations.
- We describe how we extended the statement tree of MonetDB with a *control loop node* to implement in-database query plans with shape-preserving iterations.
- We confirm the feasibility of our approach by implementing gradient descent with loops and empirically comparing the runtime of our solution with a native MonetDB implementation that flattens loops.

The paper is organized as follows. Section 2 describes the application scenario. We introduce basic terminology in Section 3. Section 4 gives an overview of related work. We introduce our approach and discuss the solution for the application scenario in Section 5. Section 6 describes the implementation in MonetDB and Section 7 evaluates our solution. We conclude in Section 8.

2 Application Scenario

Table 1 illustrates a sample data set [8] with health information of four patients. This is an extract from a right heart catheterization test data set with 63 attributes and 5735 patient records. For example, tuple `t1` in relation *rhc* refers to a patient whose PatientID[5] is 394, Age is 67.7 years, Weight is 65.9 kg, and BloodPressure is 125.0 mm Hg. The patient has not received right heart catheter treatment (value of attribute Treatment is zero) and the recovery outcome was positive (value of attribute Death is zero).

rhc

	PatientID	Age	Weight	BloodPressure	Treatment	Death
`t1`	394	67.7	65.9	125.0	0	0
`t2`	979	69.7	54.0	58.0	0	1
`t3`	1198	65.6	75.7	45.0	1	1
`t4`	4314	68.5	94.1	55.0	0	1

Fig. 1. Excerpt of the Right Heart Catheterization (`rhc`) data set

The task is to group patients into cohorts, such as each cohort consists of comparable patients. Comparable cohorts allow to compute the true effect of a treatment and

[5] We use the first character of the attribute name to refer to attributes.

decide whether a treatment is successful or not. Treatment success analysis is a common approach to predict the recovery outcome of a medication. Data sets processed in treatment success analysis typically include an additional binary feature variable that indicates whether a patient received treatment. For example, tuple `t3` in relation *rhc* represents a patient who received treatment (attribute Treatment is 1). Intuitively, one would divide the input data into a set of patients having received treatment and compare their outcome of recovery to the set of untreated patients. However, this strategy ignores any biases existing in data [12]: Among the patients there may be some, who would have recovered anyway because of their general health condition, but who still received the treatment. Similarly, it is not enough to separate patients by a recorded feature, such as gender, because groups of male and female patients might be incomparable due to other differences.

Treatment success analysis requires unbiased data, i.e., a data set containing treated and untreated patients, which according to their conditions (i.e., feature values) are comparable. Since most of data is historical, there is no possibility to randomly assign treatment to patients. Instead comparable patients must be selected deliberately to form cohorts. The propensity score represents the impact of all characteristic to a treatment and, thus, allows to match patients with similar scores. In other words, for each patient the propensity score is the probability of getting treatment based on her/his conditions. Typically the distribution into groups is done via *Propensity Score Matching*. We discuss the details in Section 5.

3 Background

We leverage the extension of SQL with relational algebra operations that supports basic matrix operations, such as multiplication and inversion, over relations [7,6]. A relation is divided into two parts as illustrated in Figure 2: Contextual information and application part. The gray cells are the contextual information, and the white cells are the application part, respectively. Contextual information identifies and describes each cell in the application part. Row and column origins are the part of the contextual information responsible for identifying and describing tuples and attributes, respectively. The application part is used in the corresponding matrix operation. Each relational matrix operation takes one or two relations with contextual information as input and delivers a result relation with row and column origins. Origins of a result relation are inherited from the contextual information of the input relation. The inheritance is based on the shape of result relation.

$t1$

P	A	B	W
394	67.7	125.0	65.9
979	69.7	58.0	54.0
1198	65.6	45.0	75.7
4314	68.5	55.0	94.1

$t2$

P	T
394	0
979	0
1198	1
4314	0

$v = \texttt{cpd}(t1, t2)$

F	T
A	65.6
B	45.0
W	75.7

Fig. 2. Input and result relations for `cpd`

Each input relation r is followed by a list of attributes $\mathbf{U}$ called order schema. The order schema is part of the contextual information and determines the order of tuples for a matrix operation. The rest of the attributes in r, $\mathcal{R} \setminus \mathbf{U}$, is called the application schema. For example, the binary relational matrix multiplication is expressed as follows:

```
SELECT * FROM MMU(r BY U, s BY V);
```

Here, r and s are input relations, and $\mathbf{U}$ and $\mathbf{V}$ are order schemas that determine the order of tuples for the multiplication.

Example 1. Consider the relational matrix crossproduct (cpd) between attributes A, B, W and attribute T from relation *rhc* sorted by values in attribute P:

```
SELECT *
FROM CPD[F]( ( SELECT P, A, B, W FROM rhc ) AS t1 BY P,
             ( SELECT P, T FROM rhc ) AS t2 BY P ) AS v;
```

Figure 2 shows the input and result relations of the crossproduct computation. Both subselects include attribute P to sort the tuples in $t1$ and $t2$ for the purpose of cpd operation. The crossproduct is performed over the values of attributes A, B, and W from relation $t1$ ordered by P and the values of attribute T from relation $t2$ ordered by P. Result relation v includes contextual information: It inherits row origin values A, B, and W from $t1$, and column origin value T from $t2$ [7]. Inheritance of these values is based on the cardinalities of the result matrix of the matrix crossproduct.

4 Related Work

Currently there are two possibilities to perform iterations inside a DBMS: to write an UDF and to use a recursive query. UDF is an expressive tool and might be used for computing iterative methods inside a database. However, in this paper we target special iterations, where both parts, i.e., iteration body and exit condition, are plain SQL expressions. When they are placed inside an UDF, they are not accessible to the optimizer, and thus, executed as-is. Unlike UDFs, we offer a solution that is deeply integrated into the system and enables optimization for both the iteration body and the exit condition.

Recursive queries are usually applied to hierarchical data in order to find a certain set of tuples. Recursive queries are based on iterations. After each iteration step, resulting tuples are added to an iterated relation until the iteration step returns an empty set of tuples. In our approach we target exit condition that is based on values in tuples of iterated relation. Since recursive queries do not preserve the shape of an iterated relation, they are not suitable for shape-preserving iterations we want to integrate into a database. Potentially, recursion can be used to perform analytical tasks, but the recursive SQL solution is not straightforward and would store all intermediate results in the iterated relation. It leads to poor time and space efficiency due to preservation of large amount of tuples that are not needed.

Jankov et al. [9] extend SimSQL database with arrays, which elements are relations, in order to bring neural networks into the relational model. Tables in an array are defined with recursive definitions, i.e., each table is defined as the result of a query over previously defined tables. Unlike our approach, each step of iteration creates a new

table. Unless materialization is stated explicitly, intermediate result tables are not materialized. Thus, space is not spent uncontrollably, but optimizer must handle enormously large query plans. The approach concentrates on how to cut plans into pieces, which are optimized and executed independently. This is an NP-hard task, which leads to some greedy heuristics used to find an approximate solution.

Binnig et al. [5] extend SQL extension with functions that support recursive iterations and multiple-tables assignment operations in order to bring interactivity and procedural flavour into SQL. New features are implemented with help of graphs with cycles. Since iterations are available only in functions, this approach does not support full integration into SELECT statement. Binnig offers a primary optimization such as push downs of selections and projections in new query graphs. However, more complicated optimization techniques, e.g., join order, are not developed yet. Optimization of cycles in trees for iterations is not discussed.

5 Propensity Score

Propensity score matching builds cohorts based on the estimation of the propensity score. Propensity score matching requires to perform the following steps: (1) computation of gradient descent between features and a target, e.g., between attributes *A*, *B*, *W* as features and attribute *T* as a target from relation *rhc*; (2) estimation of the propensity score by multiplying features and the coefficients from the gradient descent; and finally, (3) grouping of estimated propensity scores according to their similarity. We consider these steps in the following subsections.

5.1 Gradient Descent and Shape-Preserving Iterations

Gradient descent [13] is an algorithm that is often used for classification tasks, such as logistic regression. Gradient descent is an approximation method, where a cost function is iteratively minimized, while letting the coefficients converge to the optimum for the given data set [10].

In the context of relations, gradient descent takes three argument relations: The first relation includes the feature attributes (i.e., independent variables), the second relation includes an attribute that represents the target (i.e., dependent variable), the third relation includes an attribute with the initially guessed coefficients. Last relation is the result relation and is iterated over until the coefficients have converged to the real (not guessed) impact of the independent variables to the target. The iteration used in gradient descent has two key properties: (1) it is a fixed-point iteration with a cost function that must be minimized, (2) the shape of the iterated result relation remains the same (i.e., the iteration refines constant number of values, but does not change the number of attributes or tuples). We denote such iterations as *shape-preserving iterations*.

Shape-preserving iterations are used in *iterative methods* [16] from numerical analysis that refine matrices with randomly initialized values. Iterative methods are used to solve problems, for which direct methods are very expensive or do not exist, such as logistic regression.

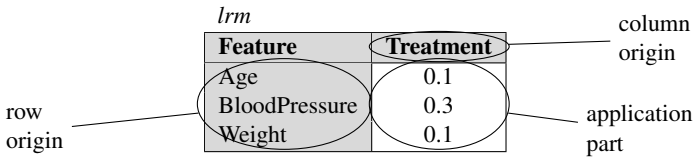


Fig. 3. Structure of an iterated result relation

The division of a relation into contextual information and an application part [6] also holds for shape-preserving iterations. Consider relation *lrm* given in Figure 3. The relation quantifies for each feature its impact on the treatment. The gray cells are the contextual information and remain unchanged during the iteration. Contextual information is composed of row and column origins and determines the shape of a relation. The white cells are the application part over which the iteration is performed.

5.2 Randomly Initialized Relations

Shape-preserving iterations require a starting point: A relation, over which they perform iterations to approximate the solution. Typically, the values in the application part of a starting point relation are generated randomly. We introduce *random relation initialization*: The mechanism for generating starting point relations for shape-preserving iterations.

Each randomly initialized relation has the structure shown in Figure 3, i.e., it includes contextual information with row and column origins and an application part. Both row origin and column origin are taken from the existing relations and determine the shape of a relation, i.e., the number of tuples and attributes.

Depending on the desired shape and meaning of result values in the result relation, the appropriate relational matrix operation is applied to existing relations. The operation provides contextual information, i.e., a skeleton for the application part. The values in the application part are generated independently of the operation result. The relational matrix algebra [7] includes an extensive set of operations, that provides all possible shapes for randomly initialized relations. For example, if the randomly initialized relation v inherits the row origin from the application schema of r and the column origin from the application schema of s, operation cpd(r, s) yields the contextual information for v.

Relation *lrm* in Figure 3 is the starting point for the gradient descent. Relation *lrm* is created as follows (see also Figure 2):

```
1  CREATE TABLE lrm(F, T) AS (
2      SELECT F, uniform [0, 1] AS T
3      FROM CPD[F]( ( SELECT P, A, B, W FROM rhc ) AS t1 BY P,
4                   ( SELECT P, T FROM rhc ) AS t2 BY P
5                 );
```

Relation *lrm* inherits its contextual information from relations $t1$ and $t2$. The shape and contextual information of *lrm* is determined by operation cpd. The row origin of *lrm* are the values of attribute F. By definition of cpd this is the application schema of $t1$, i.e. names of attributes A, B, and W. Column origin of *lrm* is attribute T. It is

inherited from application schema of $t2$. The row origin includes the feature for each coefficient. The application part of *lrm* (i.e., values of attribute T) includes random values uniformly distributed between 0 and 1.

5.3 Iterations in SQL

In order to support shape-preserving iterations we extend SQL with syntax in the `WITH` clause. Figure 4 illustrates the syntactic construction for iterations.

```
1  WITH
2      ITERATED r(T1, T2) AS (Q) UNTIL P
3  SELECT * FROM r;
```

Fig. 4. Iterations in SQL

The values in relation *r* are updated after each iteration step. Query `Q` computes the new values, and `P` is the predicate that specifies the exit condition. Thus, `Q` is repeated and relation *r* is updated until `P` evaluates to true.

Since we target shape-preserving iterations, query `Q` must not change origins of relation *r*. Input and output relations are relations with the same size and contextual information (e.g. row and column origins are kept) but with different values in the application part.

Iterations in our application scenario: To compute propensity score we perform gradient descent between attributes A, B, and W as features and attribute T as a target from relation *rhc*. Since attribute T includes binary values, gradient descent is based on standard logistic regression.

Figure 5 illustrates the SQL statement for gradient descent expressed with iterations and relational matrix algebra operations. It corresponds to classical gradient descent algorithm [13], where α is the stepsize, and `t` is the threshold. The iterative part of the query is framed and consists of `Q` and `P`. Subquery `Q` corresponds to the iteration body of the gradient descent. `cpd` is performed on lines 14-21 between features from relation *rhc* and the normalized error from relation *d* yielding the result relation $g(F,T)$. Attribute $g.T$ is the gradient corresponding to the current coefficients. Line 11 states how the coefficients in *lrm* are updated after each iteration: The gradient multiplied by the stepsize α is subtracted from the current coefficients. Subquery `P` corresponds to the exit condition. It determines if the cost function calculated in the `SUM` aggregation between the real target values $rhc.T$ and the estimated target values $e.T$ is below the given threshold t. The relational matrix operations are highlighted with red color, and the new iterative structure is highlighted with green color.

The statement iterates over a randomly initialized relation *lrm* and returns result relation *lrm*. The values in attribute F remain unchanged, and the values in attribute T are refined during the computation. Expression 1/(1+1/EXP(T)) corresponds to sigmoid function $\frac{1}{1+e^{-T}}$. Since sigmoid function takes real values and maps them into a range from zero to one, it is used in gradient descents for logistic regressions.

```
WITH
    prhc(A, B, W, P) AS (
        SELECT A, B, W, P
        FROM rhc ),
    e(P, T) AS (
        SELECT P, 1/(1+1/EXP(T)) AS T
        FROM MMU ( prhc BY P,
                   lrm BY F )
    ),
    ITERATED lrm(F, T) AS (
        SELECT lrm.F, lrm.T - α*g.T
        FROM lrm
        JOIN
        CPD[F]( prhc BY P,
                ( SELECT t1.P, t1.T/t2.N AS T
                  FROM ( SELECT rhc.P, rhc.T - e.T AS T
                         FROM rhc JOIN e
                         ON rhc.P = e.P ) AS t1,
                       ( SELECT COUNT(*) AS N
                         FROM rhc ) AS t2
                ) AS d BY P
              ) AS g ON lrm.F = g.F                          Q
    )
    UNTIL
        ( SELECT SUM(-rhc.T*log(e.T)-(1-rhc.T)*log(1-e.T))
        FROM rhc JOIN e ON rhc.P = e.P )                     P1
          <
        ( SELECT t * COUNT(*)
        FROM rhc )                                           P2
                                                                    P
SELECT * FROM lrm;
```

Fig. 5. Gradient descent over relation *rhc*

Input (lrm_0), intermediate (lrm_{50}, lrm_{100}, lrm_{200}), and output (lrm_{266}) relations are shown in Figure 6. The subscript denotes the iteration step in which the relation is computed. The intermediate relations show how the coefficients converge during the gradient descent. For example, coefficient for W converges in the beginning of the iterations, and coefficient for A converges towards the end of computation.

lrm_0

F	T
A	0.1
B	0.3
W	0.1

lrm_{50}

F	T
A	0.00
B	-0.06
W	0.03

lrm_{100}

F	T
A	0.03
B	-0.10
W	0.03

lrm_{200}

F	T
A	0.08
B	-0.16
W	0.03

lrm_{266}

F	T
A	0.11
B	-0.20
W	0.03

Fig. 6. Gradient descent: Input, intermediate, and output relations

Flexibility: Shape-preserving iterations are general and make the approach flexible. We show this for L2 regularization [11]. Regularization techniques are used to avoid overfitting, such that noise in data does not affect the result. To achieve this the formula of how the coefficients are adjusted in each iteration must be modified. For example,

in order to introduce L2 regularization into the query from Figure 5, the iteration step should be changed as follows:

```
1    SELECT lrm.F, lrm.T - α*(g.T + 2*lrm.T)
```

5.4 Estimation and Matching

The propensity score is estimated by multiplying the features with the result of the gradient descent. After estimation, groups of propensity scores are formed. Grouping data points into percentiles of propensity scores reduces bias and allows for a proper treatment success analysis [12]. Forming groups of patients with similar propensity scores is accomplished by matching tuples. There exist several approaches to perform the matching, such as stratification matching and caliper matching [4,14]. For stratification matching, the range of propensity scores is split into equally sized buckets, and each tuple is assigned to a bucket. Each bucket holds comparable tuples with treated and untreated patients.

Estimation and matching in our application scenario: To estimate the propensity score, we compute the relational matrix multiplication between features in *rhc* and coefficients in result relation *lrm*:

```
1       SELECT P, 1/(1+1/EXP(T)) AS S
2       FROM MMU ( ( SELECT A, B, W, P
3                    FROM rhc ) AS t BY P,
4                    lrm BY F )
```

Figure 7 shows the result of the multiplication: Relation *propensity_score* with attribute S being an estimated for treatment, i.e., a propensity score.

propensity_score

P	S
394	0.000
979	0.091
1198	0.617
4314	0.345

stratification_matching

P	I	S	T
394	0	0.000	0
979	0	0.091	0
1198	3	0.617	1
4314	1	0.345	0

Fig. 7. Propensity score: Estimation and matching

After the propensity score is calculated, a stratified matching is performed over the estimated scores. Here, the propensity scores are distributed into equally sized buckets with size 0.2:

```
1       SELECT P, CAST(S * 10 / 2 AS INT) AS I, S, T
2       FROM propensity_score NATURAL JOIN rhc;
```

Figure 7 illustrates the final result of propensity score matching with the stratification approach: Relation *stratification_matching* where attribute *I* is the bucket id.

6 System Implementation in MonetDB

In this section we discuss the integration of shape-preserving iterations into MonetDB.

MonetDB MonetDB is a column-store DBMS, which offers several routines optimized for column-oriented operations. MonetDB stores attributes of relations in *binary association tables* (BAT). A BAT consists of two arrays: One stores the attribute values and the other the object identifier (OID) for each tuple. Each relational operation is represented and executed as a sequence of BAT operations. The set of all BAT operations is called BAT Algebra. When an SQL query is submitted, MonetDB parses and optimizes it and builds the *statement tree*, in which each node refers to one or more attributes, or the result of an operation on attributes. The statement tree is interpreted as a sequence of BAT operations, optimized, and executed.

6.1 New Types of Nodes and Edges

We fully integrate shape-preserving iterations into MonetDB, so that a query with iterations goes through all stages of query processing.

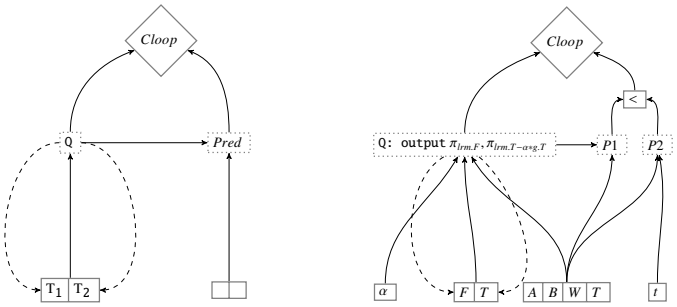
The existing structure of a statement tree is a DAG and does not provide functionality to express shape-preserving iterations. An extension is required to support cycles with an exit condition.

We introduce a new node type and a relationship. A *cloop* node is a *control* node that controls the flows in the statement tree. A cloop node models a loop with an exit condition. A *pred* node contains the predicate from the exit condition and returns a boolean instead of a BAT. The *update* relationship is controlled by a cloop node and substitutes one node with the result of another.

Consider the statement tree with the cloop node in Figure 8a. It illustrates the general shape-preserving iteration from Figure 4. A cloop node is a repeat-until structure expressed in terms of a statement tree. It has two subtrees. The left and right statement trees represent query `Q` and predicate `P`, respectively. The output of the left subtree is connected with the input nodes of this subtree by an update relationship. The right subtree has a pred node as root that contains the predicate of an exit condition and returns the result of the evaluation of this predicate. After the left and right subtrees of a cloop node have been traversed and evaluated, the cloop node decides how to proceed based on the result of the right subtree.

Example 2. Consider the SQL query in Figure 5 and the corresponding statement tree in Figure 8b. This query has a subquery `Q`: the part between AS and UNTIL keywords, and a predicate `P`: the part after the UNTIL keyword.

The root of the tree is a cloop node. The left subtree of the cloop node represents the iteration body, i.e., query `Q`. Its root node is connected by update relationship (dashed arrow) with two leaf nodes: F and T attributes from relation *lrm*. F and T are updated with $\pi_{lrm.F}$ and $\pi_{lrm.T-\alpha*g.T}$, respectively, when the predicate evaluates to false. The right subtree with root node $<$, corresponds to predicate $P1 < P2$. It evaluates the exit condition and returns a boolean.



(a) General (b) Example

Fig. 8. Statement trees with cloop node

6.2 Implementation

We implement gradient descent as a new node in a statement tree and a BAT operation in MonetDB. We compare our implementation with a native MonetDB implementation where the number of iterations is predefined and iterations are flattened in the statement tree.

Node implementation In the Node implementation we add a gradient descent node to a statement tree and a BAT operation with the gradient descent algorithm.

Algorithm 1: Gradient Descent

Input: BAT C (coefficients, initial guess), list of BATs **A** = (A_1, A_2, ...) (feature vectors), BAT T (target vector), double α (stepsize), double t (error threshold)
Output: BAT C (optimized coefficient vector)

```
1  n ← T.length()
2  repeat
3  |  E ← fill(0, n)
4  |  G ← []
5  |  foreach A_i ∈ A, val ∈C do
6  |  |  E ← E + val· A_i
7  |  end
8  |  D ← (sigmoid(E) − T)/n
9  |  foreach A_i ∈ A do
10 |  |  append(G, A_i· D )
11 |  end
12 |  C ← C - α· G
13 until cost(C, A, T) < t;
14 return C
```

Algorithm 1 illustrates the gradient descent applied to BATs and our implementation of a new BAT operation. The algorithm takes as an input BAT C with the initial

coefficients, list of feature BATs **A**, target BAT T, gradient descent stepsize α, and the error threshold t. The cost function in Equation 1 is the function we minimize with the gradient descent algorithm.

$$\texttt{cost}(\mathrm{C}, \mathbf{A}, \mathrm{T}) = (-\mathrm{T} \cdot \log(\texttt{sigmoid}(\mathbf{A} \cdot \mathrm{C})) - (1 - \mathrm{T}) \cdot \log(1 - \texttt{sigmoid}(\mathbf{A} \cdot \mathrm{C})))/n \quad (1)$$

The algorithm returns BAT C with the final coefficients as soon as the exit condition is reached, i.e., the error is smaller than the threshold.

First, BAT E, which is the non-normalized estimation of target T, is filled with zeros and then lines 5-7 are executed with the current coefficients C. Second, BAT D with the difference of the normalized estimation and the actual target is computed on line 8. BAT G with the current gradient is calculated on lines 9-11. The coefficient BAT C is updated on line 12.

Example 3. Consider Figure 9. It illustrates the first iteration step of Algorithm 1 performed over randomly initialized relation *lrm* shown in Figure 7 and relation *rhc*.

C
0.1
0.3
0.1

A_1	A_2	A_3
67.7	125.0	65.9
69.7	58.0	54.0
65.6	45.0	75.7
68.5	55.0	94.1

T
0
0
1
0

E
50.86
29.77
27.63
32.76

sigmoid(E)
1
1
1
1

D
0.25
0.25
0
0.25

G
51.45
59.50
53.50

C
0.05
0.24
0.05

Fig. 9. The first step of iteration

BAT C corresponds to attribute T from relation *lrm*, BAT list $\mathbf{A} = (\mathrm{A}_1, \mathrm{A}_2, \mathrm{A}_3)$ corresponds to attributes (A, B, W) from relation *rhc*, BAT T corresponds to attribute T from relation *rhc*, stepsize α is 0.001, and threshold t is 0.25. BAT E (i.e., the current estimation of the target) is calculated by multiplying **A** with C, and the sigmoid function is applied to E. After that BAT D is calculated between real target T and its estimation *sigmoid*(E). In our case the estimation is close to the real values only for the third patient. Then, each feature is multiplied with the normalized difference delivering gradient BAT G. Finally, BAT C is updated based on the values in G and the stepsize. After that the cost function is applied to the refined coefficients C and the until predicate is evaluated.

Native MonetDB implementation In the native MonetDB implementation Algorithm 1 is translated to a flattened statement tree, where the number of iterations is predefined and the cost function is omitted. Thus, instead of a cloop node, iteration subtrees are repeated multiple times.

The native approach has two major drawbacks. The approach does not scale, since the statement tree grows very fast. The native approach is non-robust in terms of accuracy of the result, because of the inability to access and evaluate the intermediate results during the tree creation.

7 Evaluation

Setup We extended MonetdDB v11.23.13 with the Node implementation and the native implementation. Both server and client are running on the same machine. We ran an evaluation using synthetic data on a virtual machine in the ScienceCloud [15] with Ubuntu 18.04.3 LTS, 2.593GHz Intel Haswell 4 CPU, and 16GB of RAM.

Synthetic data is generated with function `make_classification` [2], which is part of the Python library `scikit` [1]. All features in the generated data sets are informative, i.e., all features are independent variables that affect the target. All preconditions that are used for the generation of regression problems with different numbers of features and tuples are the same.

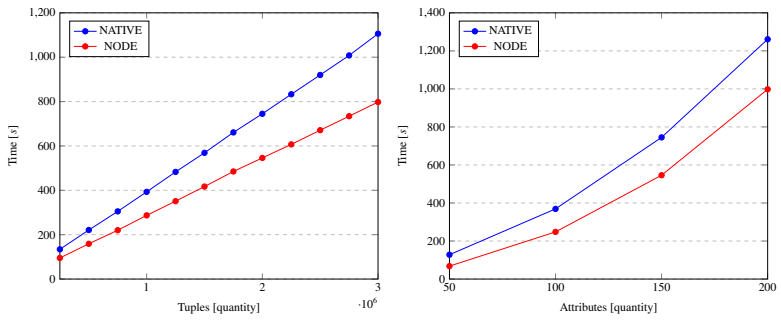


Fig. 10. Runtime of gradient descent for varying number of attributes and tuples

We perform gradient descent with the Node and native implementations over relations of different sizes. Both implementations are integrating iterations deep in MonetDB. We make sure that the Node implementation performs as many iterations as the native implementation. For the native implementation we fixed the number of iterations by passing this number within a query. For the Node implementation we set the tolerance to zero and additionally pass the maximal number of iterations. This guarantees that both approaches perform the same number of iterations.

Figure 10 illustrates the runtimes of both implementations. The left plot shows the runtimes for gradient descent applied to relations with 150 attributes (i.e., features) and a varying number of tuples. The right plot shows runtimes for relations with 2,000,000 tuples and a varying number of attributes. The Node implementation shows better performance in both cases. Note that the Node implementation computes the cost function after each iteration, while the native implementation iterates the body only. Since the native implementation flattens the statement tree, it creates huge trees with thousands of nodes, and thus, does not scale.

8 Summary

In this paper we target data that is stored in relations and that needs to be analyzed with iterative methods. We introduced shape-preserving iterations and proposed a technique

to create randomly initialized relations for the example of propensity score matching. We integrated shape-preserving iterations into SQL and the MonetDB query tree to support in-database. We illustrated the feasibility of our approach by comparing our implementation of gradient descent with a native MonetDB implementation. Future work includes the optimization of shape-preserving iterations.

References

1. scikit-learn: Machine Learning in Python. https://scikit-learn.org/, accessed: 2019-12-11
2. sklearn.datasets: Function make_classification. https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html, accessed: 2020-03-05
3. The R Stats Package: R statistical functions. https://www.rdocumentation.org/packages/stats, accessed: 2019-12-11
4. Austin, P.: Optimal caliper widths for propensity-score matching when estimating differences in means and differences in proportions in observational studies. Pharmaceutical statistics **10**, 150–61 (03 2011). https://doi.org/10.1002/pst.433
5. Binnig, C., Rehrmann, R., Faerber, F., Riewe, R.: Funsql: It is time to make sql functional. In: Proceedings of the 2012 Joint EDBT/ICDT Workshops. pp. 41–46. EDBT-ICDT '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2320765.2320786, http://doi.acm.org/10.1145/2320765.2320786
6. Dolmatova, O., Augsten, N., Böhlen, M.H.: Preserving contextual information in relational matrix operations. In: Proceedings of the 36th International Conference on Data Engineering. p. 4 pages. ICDE '20
7. Dolmatova, O., Augsten, N., Böhlen, M.H.: A relational matrix algebra and its implementation in a column store. In: International Conference on Management of Data, SIGMOD 2020, Portland, OR, USA, June 14-19, 2020. ACM (2020)
8. Frank E Harrell Jr: Right Heart Catheterization Dataset, http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/rhc.csv, accessed 2019-12-02
9. Jankov, D., Luo, S., Yuan, B., Cai, Z., Zou, J., Jermaine, C., Gao, Z.J.: Declarative recursive computation on an rdbms: Or, why you should use a database for distributed machine learning. Proc. VLDB Endow. **12**(7), 822–835 (Mar 2019). https://doi.org/10.14778/3317315.3317323, https://doi.org/10.14778/3317315.3317323
10. Martin, J.H., Jurafsky, D.: Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition. Pearson/Prentice Hall Upper Saddle River (2009)
11. Ng, A.: Feature selection, l1 vs. l2 regularization, and rotational invariance. In: Proceedings of the twenty-first international conference on Machine learning. p. 78. ACM (2004)
12. Rosenbaum, P.R., Rubin, D.B.: The central role of the propensity score in observational studies for causal effects. Biometrika **70**(1), 41–55 (1983), http://www.jstor.org/stable/2335942
13. Ruder, S.: An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747 (2016)
14. Senn, S., Graf, E., Caputo, A.: Stratification for the propensity score compared with linear regression techniques to assess the effect of treatment or exposure. Statistics in Medicine **26**(30), 5529–5544 (2007). https://doi.org/10.1002/sim.3133, https://onlinelibrary.wiley.com/doi/abs/10.1002/sim.3133
15. University of Zurich: ScienceCloud, https://www.zi.uzh.ch/en/teaching-and-research/science-it/infrastructure/, accessed 2020-03-18
16. Varga, R.S.: Matrix Iterative Analysis. Prentice-Hall Series in Automatic Computation, Prentice-Hall, Englewood Cliffs (1962)